WASHCLOTH - THE LOGICAL SUCCESSOR TO SOAPSUDS

by

Allan Gottlieb

DECEMBER 1980

REPORT NO. 029

Ultracomputer Note #12

WASHCLOTH - THE LOGICAL SUCCESSOR TO SOAPSUDS

by

Allan Gottlieb

DECEMBER 1980

REPORT NO. 029

This work was supported in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under Contract No. DE-AC02-76ER03077, and in part by the National Science Foundation under Grant No. NSF-MCS76-00116.

c.2

[illegible]

Ultracomputer Note #12
WASHCLOTH - The Logical
Successor to SOAPSUDS

Allan Gottlieb
24 December 1980

I Introduction

Simulation is a standard tool for testing the utility of a design prior to its actual implementation. In the late 60′s Schwartz [66] proposed the "Athene" class of parallel processors and Grishman implemented the SOAPSUDS simulator for such machines (Draughon etal. [67]). A dozen years later, Schwartz [80] refined his proposal to utilize the "shuffle" connection scheme, whose importance was stressed by Stone [71]. The resulting "ultracomputers" have been simulated by Gottlieb [80a & 80b] and Gottlieb and Kruskal [80].

In his more recent work, Schwartz also proposed the "paracomputer", an ideal (physically unrealizable) machine whose performance more realistic architectures could try to approach (see Gottlieb etal. [80]). The present note describes the WASHCLOTH paracomputer simulator and is organized as follows: Section II reviews the paracomputer model; section III presents the WASHCLOTH design and the machine operations provided;

section IV describes the FORTRAN interface; and section V is a users guide to WASHCLOTH.

II Paracomputers

An ideal parallel processor, dubbed a "paracomputer" by Schwartz [80], consists of identical processing elements (PEs) sharing a common memory. The individual processors may also have attached local memory which we refer to as their "private" memories; the memory shared by and common to all processors is called "public", and variables stored there are called "public variables". The PEs can simultaneously read any public cell in one cycle. Simultaneous writes are likewise effected in a single cycle and a memory cell to which such writes are directed will contain some one of the quantities written into it. This illustrates the "(paracomputer) serialization principle": If simultaneous requests are directed at the same memory cell by many PEs, the effect is as though the requests occurred in some (unspecified) serial order. Moreover, the paracomputer can effect the replace-add operation (described below) in one cycle. (These machines must be regarded as idealized computational models since physical fan-in limitations prevent their realization.)

The format of the replace-add operation is RepAdd(V,e), where V is an integer variable and e is an integer expression. This indivisible operation yields the sum S=V+e as its value and replaces the contents of storage location V by this sum S. Moreover, RepAdd satisfies the serialization principle: Assume that V is a public variable (as it ordinarily will be) and many (perhaps very many) replace-add operations simultaneously address V. Then the semantics of the replace-add operation demand that there exists some serial execution order of these replace-adds such that each operation yields the value corresponding to its position in this order, and such that V receives the the appropriate total increment. For example: If PEi executes

```
ANSi <-- RepAdd(V,ei)   ,
```

PEj simultaneously executes

```
ANSj <-- RepAdd(V,ej)   ,
```

and V is not simultaneously updated by another PEk, then the value of (the public variable) V becomes V+ei+ej, and either

```
ANSi <-- V+ei
ANSj <-- V+ei+ej
```

or

```
ANSi <-- V+ei+ej
ANSj <-- V+ej.
```

The first possibility corresponds to the following serialized order: PEi executes its replace-add, followed by PEj; the second possibility corresponds to the opposite serialization. Suppose, to be still more specific, that V initially contained the value 10, and that ei=2 and ej=6. Then after the simultaneous executions V will contain 18 and either ANSi=12 and ANSj=18 or ANSi=18 and ANSj=16.

To further illustrate this semantic rule, consider execution of the operation Repadd(I,3-I). At first glance one might assume that this is simply another way of assigning 3 to I. However, if I is a public variable it is possible for its value to change subsequent to evaluation of the expression 3-I but prior to the assignment to I triggered by the replace-add. Thus the value I returned by RepAdd could in fact be different from 3.

It is also possible to have loads, stores, and replace-adds all concurrently directed at the same memory location. Once again the serialization principle demands that the effect is as though these operations occured in some serial order. In particular, simultaneous loads from the same memory location may not yield identical results (since a simultaneous store may intervene).

III Basic Washcloth Design

WASHCLOTH simulates a paracomputer consisting of at most 48 asynchronous PEs, each of which is essentially a CDC 6600 with a few additional parallel processing instructions (most notably replace-add). Each simulated PE contains its own register set (A0,B0,X0,...,A7,B7,X7) and instruction location counter. As described below both public and private memory are simulated.

During each unit of simulated time, one instruction is executed by each PE (except when in indivisible mode; see below). Of course, different PEs may execute different instructions. By the serialization principle, concurrent operations may be simulated in any serial order. WASHCLOTH chooses to use decreasing processor number. That is, the basic operation of WASHCLOTH is to execute the following code, where #PE is the number of PE's being simulated.

```
FOR time <-- 1 TO INFINITY:
        FOR PE# <-- #PE-1 DOWNTO 0:
                {Do next instruction for this PE}
```

The instructions simulated include the full 6600 (CP, i.e. central processor) instruction set plus the following six new operations.

Name	Mnemonic	Machine Representation
Read PE Number	RPN BK	00 1 0 K *****
Indivisible	INDIV K	00 2 0 kkkkkk
Divisible	DIV K	00 3 0 kkkkkk
No Private	NOPVT	00 4 0 ? ?????
Private	PVT	00 5 0 * *****
Replace-Add	REPAD XI,BJ+K	00 I J kkkkkk

In the above table, kkkkkk is the 6 (octal) "digit" representation of K, the question marks are explained below, and a * means that the digit is ignored. (RPN, NOPVT, and PVT could have been 15 bit instead of 30 bit instructions.) We now describe the first and last of these instructions; the middle four are described below.

Read PE Number places PE# in the specified B register. Although this instruction is not essential (its effect can be obtained with a replace-add), it is a useful convenience.

The effect of

REPAD XI,BJ+K

is as follows: First the effective memory address M is obtained by adding the contents of BJ to the value of K. Then the sum S of the contents of M and the contents of XI is formed. Finally, S is stored into both M and XI. If I = 6, S is calculated using (60 bit) integer arithmetic; if I = 7 real arithmetic is used; other values of I are not permitted.

As mentioned above, concurrent instructions are serialized in decreasing order of PE#. For example, recall that register B0 is always 0 and suppose that #PE = 2. Suppose further that PE0 and PE1 concurrently execute

REPAD X6,B0+10000

with each X6 register containing 1 and (public) location 10000 containing 10. After these instructions are performed, location 10000 and PE0's X6 register contain 12, and PE1's X6 register contains 11.

The other four new instructions are needed because certain CDC operating system routines (notably those for I/O and job termination) are not compatible with the host memory management scheme used by WASHCLOTH. We now describe this scheme.

When a WASHCLOTH simulation is run, the memory of the host machine (in our case a 6600) is logically divided into 4 contiguous regions.

Low Memory

WASHCLOTH Memory

Private Memory

Public Memory

These region names are capitalized to distinguish them from simulated memory regions having similar names, e.g. WASHCLOTH allocates space for the (simulated paracomputer) public memory within the (real host) Public Memory region.

Low Memory is used by the CDC hardware, simulated programs are prohibited from storing into this area (but see the NOPVT instruction described below); the WASHCLOTH program text and program variables as well as the simulated A, B, and X registers and location counters are stored in WASHCLOTH Memory, simulated programs are prohibited from storing into this area (except in NOPVT mode); the simulated PEs private memories are allocated within Private Memory; and the paracomputer public memory is allocated within Public Memory.

The private memories are implemented in an indirect fashion due to two constraints imposed by the host machine: small address space and nonreentrant code. Since compiler generated code is not reentrant, it cannot be shared among PEs; since the address space is small, one cannot afford to replicate this code. Our solution to this problem is to place code in private memory but only replicate private memory "when proven necessary".

Management of Private Memory

Several restrictions have been imposed to guarantee that the private memory logical address space is static and the same for each PE. Specifically we assume:

1. The value of #PE is constant; there are, in effect, 48 different WASHCLOTHs, one for each possible value of #PE.

2. Every PE has an identical address space: If one PE has a variable X located at (private) location 1000, they all do; the values stored in X may, of course, differ from one PE to another. Similarly, every PE contains the same subroutines, located at the same (private) locations; when code is self modified, each PE maintains its own modification.

3. All memory is (statically) allocated at load time, i.e. there is no (370 like) GETMAIN/FREEMAIN.

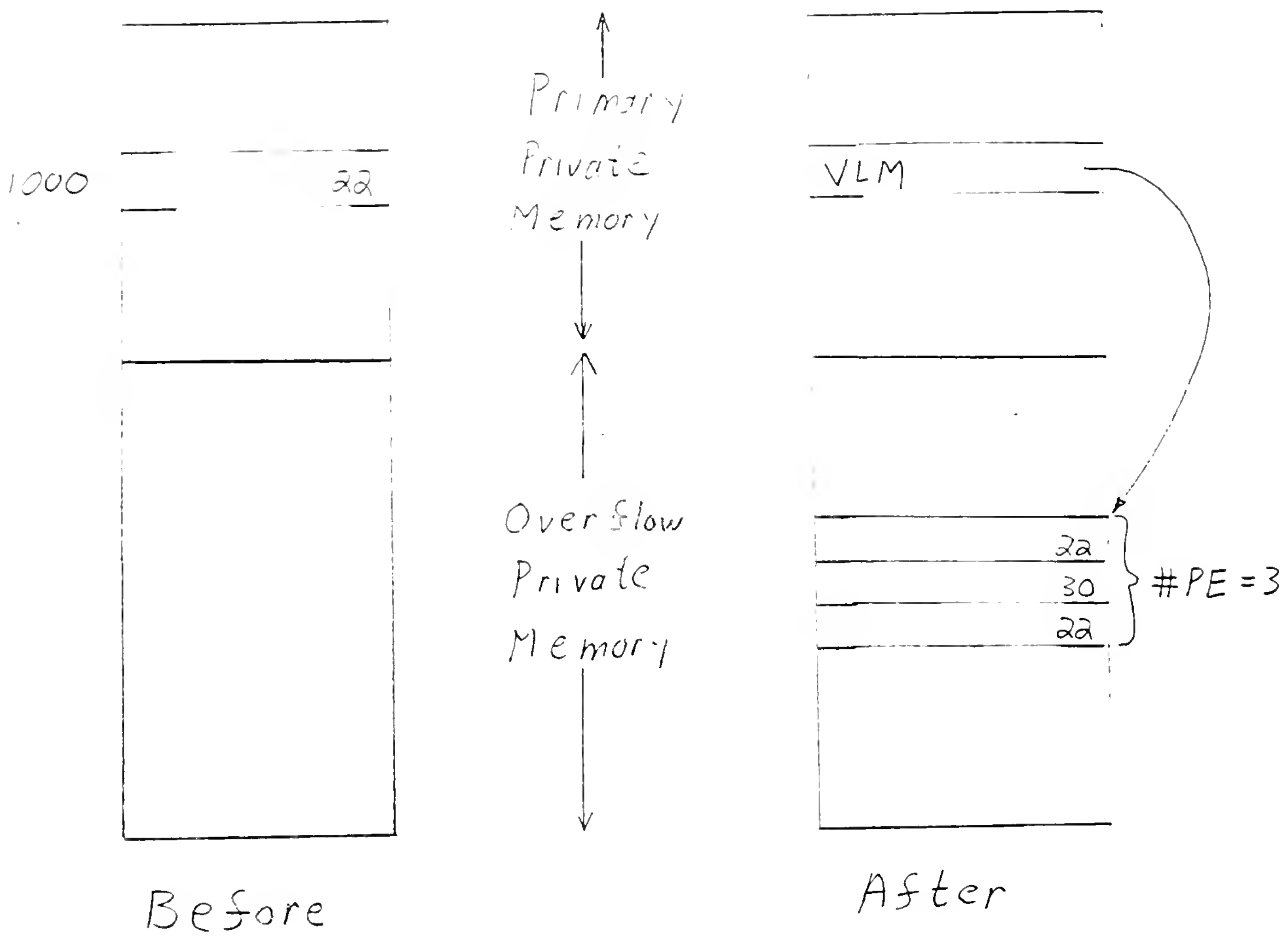
With these preliminaries taken care of, we can now understand the WASHCLOTH (private) memory management scheme. The Private Memory region mentioned above is divided into two subregions: Primary and Overflow. The Primary subregion contains one copy of the private address space. The Overflow subregion is used when two or more PEs may contain distinct values in the same private memory location.

When a simulation begins one copy of the (identical) contents of each PE's private memory is loaded into Primary Private Memory, the Overflow region is unused, and all private memory references are directed at the Primary region. This situation remains as long as no private memory locations are altered. When PEi executes a store instruction (or self modifying code, e.g. a return jump) whose target location TL is an unaltered private cell, and whose value to be stored is V, WASHCLOTH performs the following actions (see figure 1 on the next page):

A block of #PE words is obtained from the Overflow region and the value initially loaded into location TL is copied into each word of this block. (The jth word of the block, 0 <= j < #PE, will henceforth contain the value that PEj contains in location TL.) The value V is stored into PEi's location (the ith word of the block). Finally, location TL in the Primary region is given a "varying location marker" (VLM) in its high order part and a pointer to the newly created block in its low order part.

When WASHCLOTH simulates a subsequent reference to TL, it first accesses the Primary region. Upon noticing the VLM, it then follows the pointer to the Overflow block and accesses the word corresponding to the requesting PE.

Since certain CDC operating system actions are performed in part by peripheral processors, which assume a conventional CDC architecture, WASHCLOTH supplies two pairs of operations, NOPVT/PVT and INDIV/DIV that selectively disable/enable the private memory and multiprocessor facilities respectively. When



PE1 stores V = 30 into target location TL = 1000, which initially contained 22 and was not previously altered.

Figure 1

the private memory facility is disabled by a NOPVT instruction, memory references are treated in CDC fashion, i.e. no checking is done for VLM's or stores into Low or WASHCLOTH Memory. Since the accidental execution of a NOPVT instruction can cause unpredictable results, the low order 18 bits (which are not used in the instruction proper) are required to be the display code for "PVT". Thus ? ????? appearing in the table above must actually be 2 02624. The PVT command restores the private memory facility. Since issuing a NOPVT command effects all executing PEs, this command is usually used in indivisible mode. Examples of NOPVT/PVT usage are given below.

The remaining operations INDIV K and DIV K are used to specify that a block of code is to be executed indivisibly, i.e. one PE is to execute the entire block without other PE's operating. When an INDIV K request from PEi is honored, WASHCLOTH switches to indivisible mode (i.e. only instructions for PEi are simulated), which remains in effect until a DIV K is executed, at which point normal WASHCLOTH processing resumes. If a set S of PEs concurrently execute an INDIV K operation, initially the highest numbered PE in S is simulated exclusively; when this PE executes a DIV K, the next highest numbered PE in S is simulated exclusively; finally, the lowest numbered PE in S is simulated exclusively and executes a DIV K, at which point normal processing resumes.

The operands in the INDIV and DIV operations are used to specify the simulated time accrued by the indivible section. The effect of these operands is defined by: The code sequence

```
INDIV K1
{A sequence of N operations}
DIV K2
```

advances the simulated clock

$$K1 * N + K2$$

units. Although an arbitrary linear function of N can thus be specified, only two patterns have proved useful:

```
INDIV 1
{ }
DIV 0
```

in which normal timing results, and

```
INDIV 0
{ }
DIV K
```

in which the clock is incremented by K. This last pattern is appropriate when { } simulates in software a proposed hardware instruction. The current WASHCLOTH implementation restricts the operand in the INDIV instruction to be either K = 1 or K = 0, but this limitation has caused no inconvenience.

As mentioned above, the primary use of NOPVT/PVT and INDIV/DIV is to permit I/O and job termination routines to execute on the conventional CDC architecture. Let us consider some examples of this usage. Note that in these examples the I/O and termination routines are executed in NOPVT INDIV mode but all references to private variable are executed in PVT mode. To print a public variable P, the following code could be used

```
INDIV 1
NOPVT
{print P}
PVT
DIV 0
```

If P is private, the above sequence is in error since, rather than accessing P as desired, the print routine will access the (VLM, pointer) word discussed in the memory management discription given above. The following code should be used instead.

```
INDIV 1
{copy P to a public variable Q}
NOPVT
{print Q}
PVT
INDIV 0
```

Similarly, to read into a private variable P, one could write:

```
INDIV 1

NOPVT

{read Q, a public variable}

PVT

{copy Q to P}

DIV 0
```

Finally, to end execution one could write

```
INDIV 1

NOPVT

{exit routine}
```

IV The FORTRAN Interface

This section describes seven FORTRAN callable routines that invoke the new WASHCLOTH instructions and one FORTRAN callable routine that loops forever; discusses the use of (blank) COMMON for the allocation of private and public variables; presents a partial description of the CALL to WASHCLOTH; and exhibits the full text of a sample code. A complete description of the WASHCLOTH CALL as well as some debugging hints are presented in section V, the WASHCLOTH user's guide.

FORTRAN Callable Routines

The following seven routines (item 4 below comprises 2 routines) enable a FORTRAN program to use the new instructions provided by WASHCLOTH.

1. IRPN(DUMMY) -- This INTEGER FUNCTION of one (ignored) argument, returns the PE#.

2. INDIV (e) -- This SUBROUTINE with one argument, an integer expression e (which must equal 0 or 1), generates an INDIV e instruction.

3. DIV (e) -- This SUBROUTINE with one argument, an integer expression e, generates a DIV e instruction.

4. NOPVT

   PVT -- These argument-free SUBROUTINEs generate the corresponding machine instructions.

5. IREPAD(Y,e) -- This INTEGER FUNCTION of two arguments, an integer variable Y, and an integer expression e, loads the value of e into the X6 register, loads the address of Y into the B1 register, and finally executes

```
REPAD X6,B1+0
```

6. REPAD(Y,e) -- This REAL FUNCTION of two arguments, a real variable Y, and a real expression e, loads the value of e into the X7 register, loads the address of Y into the B1 register, and finally executes

```
REPAD X7,B1+0
```

We have occasionally found it convenient to have several PEs loop "forever" (e.g. to wait for a "master PE" to terminate execution of the entire job). Since the FTN compiler objects to simple infinite loops, WASHCLOTH supplies a subroutine INLOOP for this purpose.

Blank COMMON and Public Variables

Recall that the host memory is logically divided into 4 contiguous regions: Low, WASHCLOTH, Private, and Public; and that Private Memory is subdivided into Primary and Overflow components. The resulting configuration is

Low Memory

WASHCLOTH Memory

Primary Private Memory

Overflow Private Memory

Public Memory

The last 2 of these five components normally comprise (blank) COMMON, which the CDC loader automatically places in high core. Programs simulated by WASHCLOTH typically contain the following declarations:

```
        COMMON AVAIL(k), BOTPUB, public-variable-list
```

where k is some (large) integer, and public-variable-list is a list of variables. The Overflow Private Memory region starts at AVAIL and ends at BOTPUB where the Public Memory begins. Thus all the variables appearing in the public-variable-list are stored in Public Memory and thereby become public variables.

Calling Washcloth

A FORTRAN program invokes the WASHCLOTH simulator by executing a statement of the form

```
      CALL WASHCLO (AVAIL, BOTPUB, LOCF(sub)+1, UPPERT, LOWERT)
```

The first two arguments are described in the previous subsection. The third argument specifies that the simulation is to begin with each processor executing the (parameterless) subroutine sub. An exlanation of the odd syntax for this argument, a method (which has yet to be used) for beginning execution at a statement label, and the meaning of the last two (debugging) arguments are presented in section V, the WASHCLOTH user′s guide. Since the compiler must be informed that sub is a subroutine (and not a variable), an EXTERNAL statement is needed (see the example below).

Parallel Integration

Consider the problem of numerically integrating a function F defined on the interval [0,1]. One method is to subdivide the interval and compute the sum

SIGMA wi F(mi) ,

where wi and mi are the width and midpoint of the ith subinterval respectively. By choosing N equal size subintervals, this sum becomes

1/N SIGMA F((i-.5)/N) .

To parallelize this solution, we assign each PE a subset of the mi's and use the (real arithmetic) replace-add operation to compute the (public) sum.

The following simple code was prepared by B. Lubachevsky to exercise the simulator. He has subsequently simulated several significantly more complex codes including radiation transport studies and Monte Carlo analyses of physical properties. Lubachevsky's code has been revised to increase its expository value.

```
      PROGRAM SIMPLE (INPUT, OUTPUT)
C
C  THE FOLLOWING (BLANK) COMMON DECLARATION ESTABLISHES THE SIZE
C  OF OVERFLOW PRIVATE MEMORY (600 - AN EXCESSIVE VALUE),
C  THE PUBLIC VARIABLES (ALL THOSE FOLLOWING UPPERT),
C  AND THE PUBLIC VARIABLES TO BE TRAPPED (NONE).
      COMMON AVAIL(600), BOTPUB, LOWERT, UPPERT,
     *       NUMPE, NUMINT, INTPERP, DONEPES, SUM, PRINTR
      INTEGER NUMPE, NUMINT, INTPERP, DONEPES
      REAL SUM, PRINTR
C
C  VARIABLE  USE
C  --------  ---
C  NUMPE     NUMBER OF PES
C  NUMINT    NUMBER OF INTERVALS (A MULTIPLE OF NUMPE)
C  INTPERP   NUMBER OF INTERVALS PER PE
C  DONEPES   NUMBER OF PES THAT HAVE FINISHED
C  SUM       THE SUM (INTEGRAL) BEING COMPUTED
C  PRINTR    A REAL VALUE TO BE PRINTED (SEE RPRINT ROUTINE)
C
C
C  THE FOLLOWING (REQUIRED) STATEMENT INFORMES THE COMPILER
C  THAT INTEGRA IS A SUBROUTINE (AND NOT A VARIABLE).
C
      EXTERNAL INTEGRA
C
C  INITIALIZE SEVERAL PUBLIC CONSTANTS AND VARIABLES.
      NUMPE = 8
      NUMINT = 1024
      INTPERP = NUMINT / NUMPE
      DONEPES = 0
      SUM = 0.
C
C  START SIMULATION AT INTEGRATION SUBROUTINE
      CALL WASHCLO (AVAIL, BOTPUB, LOCF(INTEGRA)+1, LOWERT, UPPERT)
      END
      SUBROUTINE INTEGRA
      COMMON AVAIL(600), BOTPUB, LOWERT, UPPERT,
     *       NUMPE, NUMINT, INTPERP, DONEPES, SUM, PRINTR
      INTEGER NUMPE, NUMINT, INTPERP, DONEPES
      REAL SUM, PRINTR
      INTEGER PENUM, FRSTINT, LASTINT, IRPN, I, IREPAD
      REAL REPAD, F
C
C  DETERMINE THE SUBINTERVALS ASSIGNED TO THIS PE
      PENUM = IRPN(DUMMY)
      FRSTINT = PENUM * INTPERP + 1
      LASTINT = (PENUM + 1) * INTPERP
C
C  COMPUTE THE SUM
      DO 10 I = FRSTINT, LASTINT
         DUMMY = REPAD (SUM, F((FLOAT(I)-.5)/FLOAT(NUMINT)))
   10 CONTINUE
C
C  THE LAST PE TO COMPLETE DIVIDES BY NUMINT, PRINTS SUM,
C  AND EXITS.  ALL OTHER PES WAIT WHEN DONE.
      IF (IREPAD(DONEPES,1) .LT. NUMPE)  CALL INLOOP
      CALL RPRINT (SUM / FLOAT(NUMINT))
      CALL MYEXIT
      END


      REAL FUNCTION F(X)
      REAL X
C  TRIVIAL VERSION (IDENTITY FUNCTION)
      F = X
      RETURN
      END
      SUBROUTINE RPRINT(X)
      REAL X
      COMMON AVAIL(600), BOTPUB, LOWERT, UPPERT,
     *       NUMPE, NUMINT, INTPERP, DONEPES, SUM, PRINTR
      INTEGER NUMPE, NUMINT, INTPERP, DONEPES
      REAL SUM, PRINTR
C  ROUTINE TO PRINT A REAL VALUE.  FIRST MOVE TO PUBLIC THEN PRINT
      CALL INDIV (1)
      PRINTR = X
      CALL NOPVT
      PRINT *, PRINTR
      CALL PVT
      CALL DIV (0)
      RETURN
      END


      SUBROUTINE MYEXIT
      CALL INDIV (1)
      CALL NOPVT
      CALL EXIT
      END
```

V WASHCLOTH User's Guide

This section describes, in a terse somewhat dry manner, how to use the WASHCLOTH simulator. Many of these points are also discussed in the preceeding two sections where the internals of WASHCLOTH are exposed; here we present an outsider's view written for a (COMPASS or FORTRAN) programmer who (has read the preceeding sections and) intends to use the simulator. A FORTRAN programmer unfamiliar with COMPASS, who encounters strange terminology, should take heart -- a FORTRAN-like explanation will follow immediately.

This user's guide is divided into subsections that discuss:

1. How to invoke the WASHCLOTH simulator.

2. The JCL required.

3. The new instructions provided.

4. Some debugging hints and error messages.

5. A program developement methodology and an assessment of parallel programming with WASHCLOTH.

Invoking WASHCLOTH

The WASHCLOTH simulator, a 1000 line COMPASS program, is invoked via the usual CDC calling sequence: An RJ WASHCLO is given with register A1 pointing to a block of pointers to the five arguments. We describe these arguments in turn (see figure 2 on the next page).

Argument one is the first word (i.e. the lowest location) of the Overflow Private Memory region (called AVAIL in the WASHCLOTH annotated program text). This region must be large enough to accomodate the tables (each of size #PE) allocated when a location in private memory is updated for the first time. Argument two (called BOTPUB in WASHCLOTH) is the first word of Public Memory (which has no explicit upper bound), and is also used to terminate Overflow Private Memory. Argument three points to the first instruction to be simulated (in all PEs); the extra level of indirection is needed by FORTRAN programs. Argument four is the first word of Public Memory for which stores are to be trapped (trapping is described in the debugging subsection below). Finally, argument five is the last word of Public Memory for which stores are to be trapped.

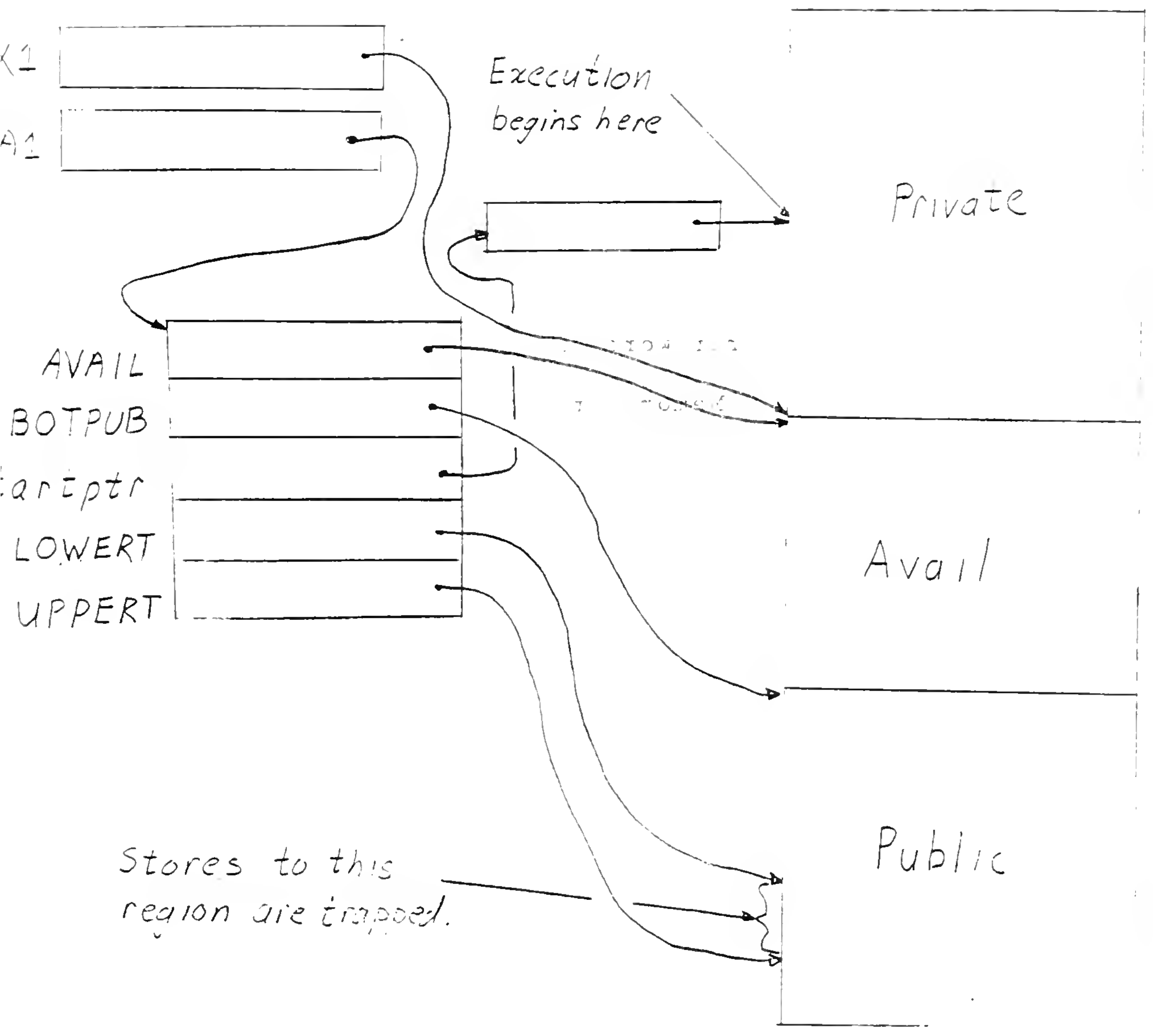
The FORTRAN program would include either

```
     CALL WASHCLO (AVAIL, BOTPUB, LOCF(sub)+1, LOWERT, UPPERT)
```

or

```
     CALL WASHCLO (AVAIL, BOTPUB, lab, LOWERT, UPPERT)
```

In the first example sub is the name of a parameterless



X1
A1
Execution begins here
Private
AVAIL
BOTPUB
tartptr
LOWERT
UPPERT
Avail
Public
Stores to this region are trapped.

Figure 2

subroutine (which has been declared EXTERNAL in the routine containing the call to WASHCLO), the LOCF built-in function returns the subroutine's entry point, and the addition of one is (COMPASS-related) magic; simulation begins at the first executable statement of sub. In the second example, lab is in an integer variable to which a label has been ASSIGNed; simulation begins at this label. To ensure that AVAIL and BOTPUB are in high core, the program would include the statement

COMMON AVAIL(k), BOTPUB, public-variables

The optimal value of k (i.e. the size of Overflow Private Memory) is very hard to determine in advance, trial and error has been our usual approach. If k is too small, a run time diagnostic is given (see debugging subsection below); if too large, space is wasted.

JCL for WASHCLOTH

The JCL required for a simulation of FORTRAN routines is simply

```
GET(W=WASHnB,T=TRAPB/UN=GOTTLBA)
FTN(OPT=0)
LOAD(W,T)
LGO.
E-O-R
FORTRAN routines
E-O-R
data
E-O-I
```

The object module WASHnB, where n = #PE = 1,2,3,...,48, contains the assembled version of the simulator proper and the FORTRAN interface. Although COMPASS programs do not need the latter, it is too small to worry about. At present WASHnB is available for n = 1, 2, 4, 8, 16, 32 and 48; however, any of the remaining 41 simulators can be trivially generated by assembling WASH/UN=GOTTLBA with the symbol #PE set to the desired value of n. The object module TRAPB contains the compiled version of the default trap routine. Users supplying their own trap routines should not load TRAPB. It is important to compile programs using OPT=0 since the optimizations performed at the default OPT=1 setting can cause errors when the program is run in parallel.

New Instructions

The non-CDC instructions supplied by WASHCLOTH are described in section III. In tabular form they are:

Name	Mnemonic	Machine Representation
Read PE Number	RPN BK	00 1 0 K *****
Indivisible	INDIV K	00 2 0 kkkkkk
Divisible	DIV K	00 3 0 kkkkkk
No Private	NOPVT	00 4 0 ? ?????
Private	PVT	00 5 0 * *****
Replace-Add	REPAD XI,BJ+K (I=6, integer arithmetic) (I=7, real arithmetic)	00 I J kkkkkk

The FORTRAN callable versions of these operations are described in section IV. They are invoked via:

```
IRPN(DUMMY)
CALL INDIV (i)
CALL DIV (i)
CALL NOPVT
CALL PVT
IREPAD(I,i)
REPAD(R,r)
```

where I and i (resp. R and r) denote an integer (resp. real) variable and expression respectively and DUMMY is ignored. An infinite loop may be obtained via

```
CALL INLOOP
```

Debugging Hints

WASHCLOTH monitors five exceptional conditions

1. Insufficient space in Overflow Private Memory.
2. Attempted store into Low or WASHCLOTH Memory.
3. User execution of a PS (program stop) instruction.
4. Invalid NOPVT instruction.
5. Attempted store into trap protected Public Memory.

The first four of these conditions are fatal errors whereas the forth is a user invoked debugging aid. After possible causes and remedies are discussed for the errors, the debugging aid is described in detail.

We have always remedied insufficient space in the Overflow Private Memory region by increasing the declared dimension of AVAIL (and thus the size of Overflow Private Memory). In more substantial applications, host memory limitations may prevent this simplistic solution. Another possible, but more difficult, corrective action is to reduce the number of private variables (or their dimensions). As with serial programs this can be accomplished by sharing space via EQUIVALENCE and labeled COMMON statements (variables declared in labeled COMMON blocks are private). Of course such coding techniques are well known to be error prone. It might also be possible to reduce #PE (at the cost of some potential parallelism).

Storing into Low Memory has always been caused by inadvertently executing system I/O or termination routines in normal WASHCLOTH mode. The remedy is to use code patterned after the examples at the end of section III, thus executing the system routines in NOPriVaTe INDIVisible mode.

A user executed PS or an invalid INDIV instruction (i.e. one in which the low order 18 bits is not the display code for "PVT") has always indicated either overwriting code with data or branching into data. In addition to checking the usual (i.e. sequential) causes for these bugs, the user is again advised to check for I/O and termination performed outside of NOPriVaTe

INDIVisible mode.

When a correct serial program is parallelized, errors often arise in synchronizing the use of shared resources. These errors are particularly difficult to isolate by traditional methods since the addition of debugging statements can cause previously simultaneous actions no longer to be concurrent. The user needs to obtain diagnostic information (generally concerning the use of shared resources) without affecting concurrencies. The WASHCLOTH trapping facility is designed to satisfy this need.

The last two arguments passed to WASHCLOTH, LOWERT and UPPERT, delimit a region of Public Memory to which all stores are trapped. The user specifies which (public) program variables are included within this region by a (blank) COMMON declaration, those variables appearing between LOWERT and UPPERT inclusive are monitored for store traps.

```
COMMON AVAIL(k),BOTPUB,...,LOWERT,trapped-area,UPPERT,...
```

Whenever a monitored variable is updated, a trap occurs for which the default action is to print a diagnostic containing

1. The number of the PE involved.
2. The current value of the simulated clock.
3. The (host) machine location affected.
4. The value in this location before the update.
5. The value in this location after the update.

The address is printed in octal and the before and after values are each printed in octal, decimal, and exponential notations. Thus a total of nine numbers are printed.

Several points should be noted: If many PEs update the same monitored variable, a trap occurs for each PE; if these updates are simultaneous, the traps are serialized in decreasing order of PE#. Since execution of a trap does not effect the simulated time taken by an operation, concurrencies are preserved. Finally, to avoid the large volume of printed output that may result from indiscriminant trapping, the above default may be easily overridden in the following manner.

The diagnostic described above is produced by a trivial FORTRAN routine consisting of just two executable statements: PRINT and RETURN. Thus to customize the trapping actions, the user need only rewrite this one routine, which has a calling sequence (equivalent to)

```
        CALL TRAP (pen, clock, loc, oldval, newval)
```

Since WASHCLOTH invokes TRAP, the later is executed directly by the 6600 and is <u>not</u> simulated by WASHCLOTH. Thus TRAP may be writted as a conventional (serial) program (e.g. the NOPVT/PVT and INDIV/DIV operators are not used).

A user supplied trap routine would likely be significantly more sophisticated than the trivial default routine described above: Instead of printing diagnostic information each trap, the user may wish to compute summary data, which is then printed periodically (e.g. after every nth trap). A trap routine can access all public variables by simply including a (blank) COMMON declaration. To examine private variables, however, is more difficult (due to the indirect storage management scheme used for private variables) and would be most naturally implemented in COMPASS using the folling scheme: If the location being trapped does not contain the VLM (octal 00202211260124, the display code for ":PRIVAT") in its high order 42 bits, then this location contains the true value for all PEs. If the VLM is found, the pointer in the low order 18 bits points to a block of #PE words in which the value for PEi is contained in word i. It must be noted, however, that it has always been the use of shared resources involving public variables that caused our "parallel errors".

Our Experiences

This subsection presents some ideas (largely due to B. Lubachevsky) concerning a program development methodology and an assessment of parallel programming. These remarks concern scientific applications programming and assume the existence of standard system routines for synchronization, parallel queue management, etc. (see, for example, Gottlieb etal. [80]). I must point out, however, that our comments are based on limited experience; to date we have implmented just a few dozen algorithms using fewer than 1000 individual runs.

Unfortunately, the most difficult task, finding a suitable algorithm for the problem at hand, is further complicated by the desire for high parallelizability. We have found no short cut for this effort. After the parallel algorithm has been specified, our methodology proceeds as follows:

Implement a one PE version without WASHCLOTH, i.e. implement this parallel algorithm in a serialized form. Note that the resulting serial implementation is likely to be less efficient than one based on an algorithm expressly designed to be run serially.

Then use WASHCLOTH with one PE. This introduces the private/public distinction without the added problem of simultaneity. In particular, I/O and termination require NOPVT mode and we recommend using stylized routines for these tasks (see, for example, RPRINT and MYEXIT at the end of section IV).

Finally, raise #PE to an appropriate value and use (FORMATted) I/O within the body of the routines. Synchronization problems often surface during this last stage, but we have found the trapping mechanism adequate for dealing with them.

We now roughly compare the effort required to develop and implement a parallel algorithm (using the methodology described above) with the effort required to obtain a serial solution, i.e.

to develop and implement a serial algorithm for the same problem. To date, when using the above procedure, at least 80% of the time is expended during the first two steps: developing a parallel algorithm and implementing a one PE version without WASHCLOTH. Were the goal a serial solution, this effort would be reduced (since serial algorithms are usually easier than parallel ones) and all subsequent steps would be eliminated. However, for the problems we have solved, the serial algorithms are not much easier and we believe that "going parallel" no more than doubles the effort required to implement those scientific applications that are inherently suitable for parallel programming.

References

E. Draughon, R. Grishman, J. Schwartz, and A. Stein [67], "Programming Considerations for Parallel Computers", IMM 362, Courant Institute, NYU, NY.

Allan Gottlieb [80a], "PLUS: A PL/I Based Ultracomputer Simulator, I", Ultracomputer Note #10, Courant Institute, NYU, NY.

Allan Gottlieb [80b], "PLUS: A PL/I Based Ultracomputer Simulator, II", Ultracomputer Note #14, Courant Institute, NYU, NY.

Allan Gottlieb and Clyde P. Kruskal [80], "MULT: A Multitasking Ultracomputer Language With Timing", Ultracomputer Note #15, Courant Institute, NYU, NY.

Allan Gottlieb, Boris Lubachevsky, and Larry Rudolph [80], "Efficient Techniques for Coordinating Large Numbers of Cooperating Sequential Processors", Ultracomputer Note #16, Courant Institute, NYU, NY.

J. T. Schwartz [66], "Large Parallel Computers", JACM 13, pp. 25-32.

J. T. Schwartz [80], "Ultracomputers", TOPLAS 2, pp. 484-521.